Chapitre 14

Listes chaînées

14.1 La notion de liste

Une liste est une structure de données qui permet de stocker une séquence d'objets d'un même type. En cela, les listes ressemblent aux *tableaux*. La séquence d'entiers 3, 7, 2, 4 peut être représentée à la fois sous forme de tableau ou de liste. La notation [3, 7, 2, 4] représentera la liste qui contient cette séquence.

Il y a cependant des différences fondamentales entre listes et tableaux :

- Dans un tableau on a accès immédiat à n'importe quel élément par son indice (accès dit *aléatoire*), tandis que dans une liste chaînée on a accès aux éléments un après l'autre, à partir du premier élément (accès dit *séquentiel*).
- Un tableau a une taille fixe, tandis qu'une liste peut augmenter en taille indéfiniment (on peut toujours rajouter un élément à une liste).

Définition (récursive) :

En considérant que la liste la plus simple est *la liste vide* (notée [ ]), qui ne contient aucun élément, on peut donner une définition récursive aux listes chaînées d'éléments de type `T` :

- la liste vide [ ] est une liste ;
- si e est un élément de type `T` et l est une liste d'éléments de type `T`, alors le couple (e, l) est aussi une liste, qui a comme premier élément e et dont le reste des éléments (à partir du second) forment la liste l.

Cette définition est *récursive*, car une liste est définie en fonction d'une autre liste. Une telle définition récursive est correcte, car une liste est définie en fonction d'une liste plus courte, qui contient un élément de moins. Cette définition permet de construire n'importe quelle liste en partant de la liste vide.

Conclusion : la liste chaînée est une structure de données récursive.

La validité de cette définition récursive peut être discutée d'un point de vue différent. Elle peut être exprimée sous la forme suivante : quelque soit la liste l considérée,

- soit l est la liste vide [ ],
- soit l peut être décomposée en un premier élément e et un reste r de la liste, l=(e, r).

Cette définition donne *une décomposition récursive* des listes. La condition générale d'arrêt de récursivité est pour la liste vide. Cette décomposition est valide, car la liste est réduite à chaque pas à une liste plus courte d'une unité. Cela garantit que la décomposition mène toujours à une liste de longueur 0, la liste vide (condition d'arrêt).

14.2 Représentation des listes chaînées en Java

Il y a plusieurs façons de représenter les listes chaînées en Java. L'idée de base est d'utiliser *un enchaînement de cellules* :

- chaque cellule contient un élément de la liste ;
- chaque cellule contient une référence vers la cellule suivante, sauf la dernière cellule de la liste (qui contient une référence nulle) ;
- la liste donne accès à la première cellule, le reste de la liste est accessible en passant de cellule en cellule, suivant leur enchaînement.

La figure 14.1 illustre cette représentation pour la liste exemple ci-dessus [3, 7, 2, 4].

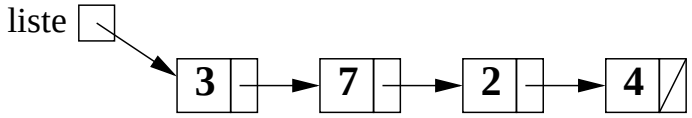

FIGURE 14.1 – Représentation par enchaînement de cellules de la liste [3, 7, 2, 4]

En Java, les références ne sont pas définies explicitement, mais implicitement à travers les objets : un objet est représenté par une référence vers la zone de mémoire qui contient ses variables d'instance.

Comme une liste est définie par une référence vers une cellule (la première de la liste), *la solution la plus simple est de représenter une liste par sa première cellule*. La classe suivante définit une liste d'entiers.

Listing 14.1 – (lien vers le code brut)

```
1 class ElementListe{
2   int valeur;
3   ElementListe suivant;
4
5   ElementListe(int valeur, ElementListe suivant){
6     this.valeur = valeur;
7     this.suivant = suivant;
8   }
9 }
```

Cependant, cette représentation ne fonctionne que pour les listes non vide (un `ElementListe` contient forcément au moins une valeur).

On représente donc d'habitude les listes par deux classes : une classe pour les éléments, et une classe pour la liste elle-même, qui contient une référence vers son premier élément.

Listing 14.2 – (lien vers le code brut)

```
1 class Liste {
2   ElementListe premier;
3 }
```

Si `premier` est à `null`, la liste est vide.

14.3 Opérations sur les listes chaînées, version itérative

Nous étudierons les opérations les plus importantes sur les listes chaînées, qui seront implémentées comme des méthodes de la classe `Liste`.

Conformément à leur définition, les listes sont des structures *récursives*. Par conséquent, les opérations sur les listes s'expriment naturellement par des algorithmes récursifs. En même temps, les listes sont des structures linéaires, parfaitement adaptées à un parcours *itératif*. Nous allons proposer d'abord une version des classes qui implémentera les méthode de manière *itérative*, et nous en donnerons ensuite une version *récursive.*

La représentation de la classe `Liste` ci-dessous montre sous forme de méthodes les opérations sur listes que nous discuterons.

Listing 14.3 – (lien vers le code brut)

```
1
2  public class Liste {
3          public boolean estVide() {}
4          public ElementListe getDebut() {}
5          public void ajouterAuDebut(int v) {}
6          public int getLongueur() {}
7          public boolean contient(int v) {}
8          public void retirerPremiereOccurrence(int v) {}
9          public void concatener(Liste l) {}
10
11 }
```

14.3.1 La classe ElementListe

Nous avons décidé de placer toute la logique des méthodes dans la classe `Liste.` Ce n'est pas forcément la seule solution, mais le code obtenu sera plus facile à expliquer.

La classe `ElementListe` est donc réduite à sa plus simple expression : un élément de liste a une *valeur,* et est suivi d'un autre élément de liste (qui peut être éventuellement `null` si notre élément est le dernier).

La classe est dotée de plusieurs constructeurs et des accesseurs nécessaires.

Listing 14.4 – (lien vers le code brut)

```
1  public class ElementListe {
2          private int valeur;
3          private ElementListe suivant;
4
5          public ElementListe(int valeur, ElementListe suivant) {
6                  this.valeur = valeur;
7                  this.suivant = suivant;
8          }
9
10         /**
11          * Crée un élément de liste sans successeur.
12          * @param v
13          */
14         public ElementListe(int v) {
15                 this.valeur = v;
16                 this.suivant = null;
17         }
18
19         public int getValeur() {
```

```
20                  return valeur;
21          }
22
23          public void setValeur(int valeur) {
24                  this.valeur = valeur;
25          }
26
27          public ElementListe getSuivant() {
28                  return suivant;
29          }
30
31          public void setSuivant(ElementListe suivant) {
32                  this.suivant = suivant;
33          }
34  }
```

Consulter la valeur d'un élément et accéder à l'élément suivant

Ces opérations sont réalisées par les méthodes `getValeur` (qui retourne la valeur associée à un élément e de la liste), respectivement `getSuivant` (qui retourne l'élément qui suit e, ou `null` si e est le dernier élément).

En rendant les variables d'instance `valeur` et `suivant` private, nous nous sommes interdit d'y accéder directement, d'où l'écriture de ces méthodes, appelées *accesseurs.*

Nous verrons plus tard que l'accès direct aux variables d'instance est déconseillé dans la programmation orientée-objet. A la place, on préferre "cacher" les variables d'instance et donner accès à leur valeurs à travers des méthodes. Ceci fera l'objet d'un prochain cours.

Remarque : Il ne faut pas oublier que `getValeur` et `getSuivant` sont des méthodes de la classe `ElementListe`, donc une instruction comme `return valeur` signifie `return this.valeur`

.

Modifier la valeur et l'élément suivant

Ces opérations sont réalisées par les méthodes `setValeur` (qui modifie la valeur d'un élément de la liste), respectivement `setSuivant` (qui change l'élément suivant).

Ces méthodes permettent donc de *modifier* les composants `valeur` et `suivant` d'une cellule de la liste. Elles sont complémentaires aux méthodes `getValeur()` et `getSuivant()`, qui permettent de *consulter* ces mêmes composants.

Leur présence dans la classe `ElementListe` correspond au même principe que celui évoqué pour `getValeur()` et `getSuivant()` : remplacer l'accès direct aux variables d'instance par des appels de méthodes.

14.3.2 Opérations sans parcours de liste

Pour ces opérations il n'y a pas de variantes itératives et récursives, l'action est réalisée directement sur l'objet `Liste`.

Obtenir le premier élément de la liste

Cette opération est réalisée par la méthode `getPremier()` qui retourne le premier `ElementListe` de la liste.

Listing 14.5 – (lien vers le code brut)

```
1   public ElementListe getPremier() {
2     return premier;
3   }
```

Savoir si une liste est vide

Une liste est vide si son premier élément est à `null`. La méthode `estVide()` de la classe `Liste` est donc :

Listing 14.6 – (lien vers le code brut)

```
1   public boolean estVide() {
2     return premier == null;
3   }
```

Insérer un élément en tête de liste

L'insertion en tête de liste est réalisée par la méthode `ajouterAuDebut`. C'est une opération simple, car elle nécessite juste un accès à la tête de la liste initiale. On crée une nouvelle cellule, à laquelle on enchaîne l'ancienne liste.

Listing 14.7 – (lien vers le code brut)

```
1   public void ajouterAuDebut(int v) {
2     ElementListe ancienPremier= premier;
3     premier= new ElementListe(v,ancienPremier);
4   }
```

14.3.3 Opérations avec parcours de liste - variante itérative

Ces opérations nécessitent un parcours complet ou partiel de la liste. Dans la variante itérative, le parcours est réalisé à l'aide d'une référence qui part de la première cellule de la liste et suit l'enchaînement des cellules.

La figure 14.2 illustre le principe du parcours itératif d'une liste à l'aide d'une référence *ref*.

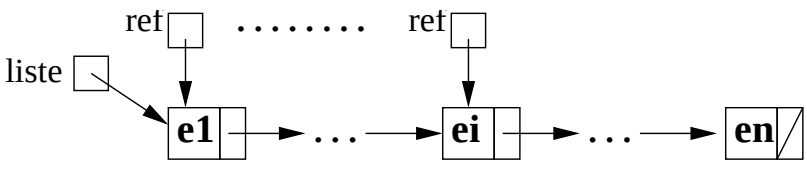

FIGURE 14.2 – Parcours itératif d'une liste à l'aide d'une référence

Calculer la longueur d'une liste

Cette opération, réalisée par la méthode `getLongueur`, nécessite un parcours complet de la liste pour compter le nombre de cellules trouvées.

Listing 14.8 – (lien vers le code brut)

```
1   public int getLongueur() {
2     int longueur= 0;
3     ElementListe ref= getPremier();
4     while (ref != null) {
5       longueur++;
6       ref= ref.getSuivant();
7     }
8     return longueur;
9   }
```

Remarque : La référence *ref* est positionnée au début sur la première cellule de la liste et avance jusqu'à ce qu'elle devient nulle à la fin de la liste. Chaque fois qu'elle pointe vers une nouvelle cellule, le compteur est incrémenté. Remarquez que l'avancement dans la liste se fait en utilisant la méthode `getSuivant` de la cellule courante.

Vérifier l'appartenance d'un élément à une liste

Cette opération, réalisée par la méthode `contient`, nécessite un parcours partiel de la liste pour chercher l'élément en question. Si l'élément est retrouvé, le parcours s'arrête à ce moment-là, sinon il continue jusqu'à la fin de la liste.

Listing 14.9 – (lien vers le code brut)

```
1   public boolean contient(int v) {
2     boolean trouve= false;
3     ElementListe ref= getPremier();
4     while (! trouve && ref != null) {
5       if (ref.getValeur() ==v ) {
6         trouve= true;
7       } else {
8         ref= ref.getSuivant();
9       }
10    }
11    // trouve est vrai implique donc ref.getValeur() ==v
12    // autre test possible pour la boucle
13    // while (ref != null && ref.getValeur() != v)
14    // expliquer l'ordre des test dans ce cas.
15    return trouve;
16  }
```

Concaténation de deux listes

Cette opération, réalisée par la méthode `concatener`, rajoute à la fin de la liste courante toutes les cellules de la liste passée en paramètre. Elle nécessite un parcours complet de la liste courante, afin de trouver sa dernière cellule.

La liste obtenue par concaténation a toujours la même première cellule que la liste initiale. On peut dire donc que la liste initiale a été modifiée, en lui rajoutant par concaténation une autre liste.

Tel que le code est écrit, les éléments de la liste `l` sont partagés entre `l` et la liste $l0$ à laquelle on l'a concaténée. Une modification ultérieure de `l` peut avoir des conséquences inattendues sur $l0$. Une variante plus sûre, mais plus coûteuse en temps et en espace mémoire serait de dupliquer le contenu de `l`.

Listing 14.10 – (lien vers le code brut)

```
1   public void concatener(Liste l) {
2     if (this.estVide()) {
3       this.premier= l.getPremier();
4     } else {
5       // On parcourt la liste jusqu'à être sur
6       // le dernier élément, celui dont le suivant est null.
7       ElementListe dernier= this.getPremier();
8       while (dernier.getSuivant() != null) {
9         dernier= dernier.getSuivant();
10      }
11      // Nous y sommes. dernier correspond au dernier élément
12      // de la liste, qui existe car celle-ci n'est pas vide.
13      // On fixe donc le suivant de ''dernier'' au premier
14      // élément de la liste l.
15      dernier.setSuivant(l.getPremier());
16    }
17  }
```

Remarque : La condition de la boucle *while* change ici, car on veut s'arrêter sur la dernière cellule et non pas la dépasser comme dans les méthodes précédentes. Remarquez que l'enchaînement des deux listes se fait en modifiant la variable d'instance `suivant` de la dernière cellule à l'aide de la méthode `setSuivant`.

Dans le même esprit que l'algorithme de concaténation, on peut écrire l'ajout d'un élément en *dernière position* d'une liste :

Listing 14.11 – (lien vers le code brut)

```
1   public void ajouterALaFin(int v) {
2     if (estVide()) {
3       premier= new ElementListe(v);
4     } else {
5       // Il y a un dernier élément.
6       // On le cherche et on ajoute après lui.
7       ElementListe dernier = getDernierElement();
8       // nous savons que
9       // dernier.getSuivant() == null => dernier est bien le dernier élément.
10      dernier.setSuivant(new ElementListe(v));
11    }
12  }
13
14  /**
15  * Trouve le dernier élément d'une liste non vide.
16  * Lance une {@link NullPointerException} pour une liste vide.
```

```
17    * @return le dernier élément d'une liste
18    * @throws une NullPointerException si la liste est vide
19    */
20   private ElementListe getDernierElement() {
21     ElementListe dernier= premier;
22     while (dernier.getSuivant() != null) {
23       dernier= dernier.getSuivant();
24     }
25     return dernier;
26   }
```

(la méthode `getDernierElement` pourra être réutilisée avec profit dans `concatener` pour simplifier son écriture).

Suppression de la première occurrence d'un élément

Cette opération, réalisée par la méthode `retirerPremiereOccurrence`, élimine de la liste la première apparition de l'élément donné en paramètre (s'il existe). On distingue comme précédemment les cas où la cellule modifiée est la première et les autres

La suppression d'une cellule de la liste se fait de la manière suivante (voir la figure 14.3) :

- si la cellule est la première de la liste, la nouvelle liste sera celle qui commence avec la seconde cellule.
- sinon la cellule a un prédécesseur ; pour éliminer la cellule il faut la "court-circuiter", en modifiant la variable `suivant` du prédécesseur pour qu'elle pointe vers la même chose que la variable `suivant` de la cellule.

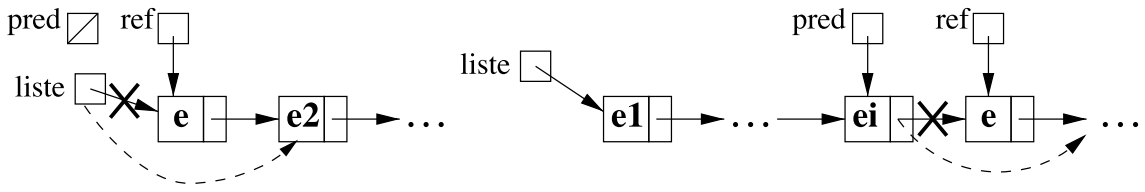

FIGURE 14.3 – Suppression d'une cellule de la liste

Le parcours de la liste à la recherche de l'élément donné a une particularité : si la référence s'arrête sur la cellule qui contient l'élément, la suppression n'est plus possible, car on n'a plus accès au prédécesseur (dans les listes chaînées on ne peut plus revenir en arrière).

La méthode utilisée est alors de parcourir la liste avec *deux références* :

- une (*elt*) qui cherche la cellule à éliminer ;
- une autre (*precedent*) qui se trouve toujours sur le prédécesseur de *elt*.

Listing 14.12 – (lien vers le code brut)

```
1    public void retirerPremiereOccurrence(int v) {
2      // On élimine le problème de la liste vide
3      if (estVide())
4            return;
5      // Le but est de trouver l'élément qui précède v...
6      // qui n'existe pas si v est la première valeur=>
7      if (premier.getValeur() == v) {
```

```
8         premier= premier.getSuivant();
9       } else {
10        ElementListe precedent= premier;
11        ElementListe elt= premier.getSuivant();
12        while (elt != null && elt.getValeur() != v) {
13          precedent= elt;
14          elt= elt.getSuivant();
15        }
16        if (elt != null) {
17          // L'élément a été trouvé
18          // Plomberie:
19          precedent.setSuivant(elt.getSuivant());
20        }
21      }
```

14.3.4 Opérations avec parcours de liste - variante récursive

Ces opérations sont basées sur une décomposition récursive de la liste en *un premier élément* et *le reste de la liste* (voir la figure 14.4). Le cas le plus simple (condition d'arrêt) est la liste avec une seule cellule.

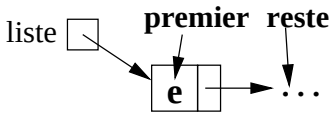

FIGURE 14.4 – Décomposition récursive d'une liste

Calcul récursif de la longueur d'une liste

La formule récursive est :

```
longueur(listeElements) = 0                    si listeElements est vide
                  1 + longueur(listeElements.suivant()) sinon
```

La fonction `getLongueur` s'écrit alors très facilement, comme suit :

Listing 14.13 – (lien vers le code brut)

```
1   public int getLongueur() {
2     return getLongueurRec(premier);
3   }
4
5
6   private int getLongueurRec(ElementListe elt) {
7     if (elt == null)
8       return 0;
9     else
10      return 1 + getLongueurRec(elt.getSuivant());
11  }
```

Dans notre exemple, la « vraie » fonction récursive est la seconde. Mais comme il n'est pas pratique pour le programmeur qui utilise la classe d'écrire

Listing 14.14 – (lien vers le code brut)

```
1   int l= maListe.getLongueurRec(maListe.getPremier());
```

on cache l'appel récursif en fournissant la méthode publique `getLongueur()` qui « amorce » l'appel, le travail étant ensuite réalisé par la méthode récursive.

Vérification récursive de l'appartenance à une liste

La formule récursive est :

```
contient(listeElements, e) = false si listeElements == null
                             true si listeElements != null
                                     et listeElements.valeur == e
                             contient(listeElements.suite, e) sinon
```

Remarquez que dans ce cas il y a deux conditions d'arrêt : quand on trouve l'élément recherché ou quand la liste est vide. La fonction récursive `contient` s'écrit alors comme suit :

Listing 14.15 – (lien vers le code brut)

```
1   public boolean contient(int v) {
2     return contient(getPremier(), v);
3   }
4
5   private boolean contientRec(ElementListe elt, int v) {
6     // Si la liste est vide, elle ne contient pas v:
7     if (elt == null)
8       return false;
9     else if (elt.getValeur()== v)
10      // Sinon, si elle commence par v, alors elle le contient
11      return true;
12    else
13      // Sinon, elle contient v si la suite de la liste le contient
14      return contientRec(elt.getSuivant(), v);
15  }
```

Ici encore, la fonction récursive est la seconde, la pemière étant une façade plus agréable d'utilisation.

Concaténation récursive de deux listes

Listing 14.16 – (lien vers le code brut)

```
1   public void concatener(Liste l) {
2     if (this.estVide()) {
3       this.premier= l.premier;
4     } else {
5       concatenerRec(premier,l.getPremier());
6     }
7   }
```

```
8
9     /**
10    * Méthode appelée uniquement si l0 est non null.
11    */
12    private void concatener(ElementListe l0, ElementListe l1) {
13      if (l0.getSuivant() == null) {
14        l0.setSuivant(l1);
15      } else {
16        concatenerRec(l0.getSuivant(), l1);
17      }
18    }
```

Suppression récursive de la première occurrence d'un élément

Pour supprimer la première occurrence d'un élément, le principe est : La formule récursive est :

```
supprimerPremier(l, e) =
       null                  si l est null
       l.suite               si l.premier == e
       ElementListe(l.premier, supprimerPremier(l.suite, e)) sinon
```

Si le premier élément de la liste est celui recherché, le résultat sera le reste de la liste. Sinon, le résultat sera une liste qui aura le même premier élément, mais dans la suite de laquelle on aura supprimé la valeur cherchée.

Listing 14.17 – (lien vers le code brut)

```
1     public void retirerPremiereOccurrence(int v) {
2       // On élimine le problème de la liste vide
3       if (! estVide()) {
4         premier= retirerPremiereOccurrenceRec(premier, v);
5       }
6     }
7
8     public ElementListe retirerPremiereOccurrenceRec(ElementListe l, int v) {
9       if (l== null) {
10        return l;
11      } else if (l.getValeur() == v) {
12        return l.getSuivant();
13      } else {
14        return new ElementListe(l.getValeur(),
15                                retirerPremiereOccurrenceRec(l.getSuivant(), v));
16      }
17    }
```

Remarque : L'inconvénient de cette méthode est qu'on crée des copies de toutes les cellules qui ne contiennent pas l'élément recherché. Pour éviter cela, la définition doit mélanger calcul et effets de bord, comme suit :

Listing 14.18 – (lien vers le code brut)

```
1     public ElementListe retirerPremiereOccurrenceRec(ElementListe l, int v) {
```

```
2       if (l== null) {
3         return l;
4       } else if (l.getValeur() == v) {
5         return l.getSuivant();
6       } else {
7         l.setSuivant(retirerPremiereOccurrenceRec(l.getSuivant(), v));
8         return l;
9       }
10    }
```

14.4 Listes triées

Nous nous intéressons maintenant aux listes chaînées dans lesquelles les éléments respectent un ordre croissant. Dans ce cas, les méthodes de recherche, d'insertion et de suppression sont différentes. L'ordre des éléments permet d'arrêter plus tôt la recherche d'un élément qui n'existe pas dans la liste. Aussi, l'insertion ne se fait plus en début de liste, mais à la place qui préserve l'ordre des éléments.

Nous utiliserons une classe `ListeTriee` qui est très similaire à la classe `Liste`, mais dans laquelle on s'intéresse uniquement aux méthodes de recherche (`contientTriee`), d'insertion (`insertionTriee`) et de suppression (`suppressionTriee`). Les autres méthodes sont identiques à celles de la classe `Liste`.

Listing 14.19 – (lien vers le code brut)

```
1  class ListeTriee{
2    ElementListe premier;
3
4    public ElementListe getPremier() {...}
5
6    public boolean contientTriee(int elem){...}
7    public void ajouterTriee(int elem){...}
8    public void retirerPremiereOccurrence(int elem){...}
9  }
```

14.4.1 Recherche dans une liste triée

La différence avec la méthode `contient` de `Liste` est que dans le parcours de la liste on peut s'arrêter dès que la cellule courante contient un élément plus grand que celui recherché. Comme tous les éléments suivants vont en ordre croissant, ils seront également plus grands que l'élément recherché.

Variante iterative :

Listing 14.20 – (lien vers le code brut)

```
1    public boolean contientTriee(int v) {
2      ElementListe elt = getPremier();
3      while (elt != null && elt.getValeur() < v) {
4        elt = elt.getSuivant();
5      }
6      return (elt != null && elt.getValeur() == v);
7    }
```

Variante récursive :

Listing 14.21 – (lien vers le code brut)

```
1     public boolean contientTriee(int v) {
2         return contientTrieeRec(getPremier(), v);
3     }
4
5     private boolean contientTrieeRec(ElementListe elt, int v) {
6         if (elt == null) {
7             return false;
8         } else if (elt.getValeur() > v) {
9             return false;
10        } else if (elt.getValeur() == v) {
11            return true;
12        } else {
13            return contientTrieeRec(elt.getSuivant(), v);
14        }
15    }
```

14.4.2 Insertion dans une liste triée

L'insertion d'un élément doit se faire *à la bonne place* dans la liste. La bonne place est *juste avant la première cellule qui contient un élément plus grand que celui à insérer*.

Variante iterative :

La figure 14.5 montre les deux cas d'insertion :

- en début de liste.
  Elle est similaire alors à l'opération `insertionDebut` de la classe `Liste`
- en milieu de liste.
  La recherche de la position d'insertion se fait alors avec une méthode similaire à celle utilisée pour la méthode `suppressionPremier` de la classe `Liste`, avec deux références. Le prédécesseur doit être modifié pour pointer vers la nouvelle cellule, tandis que celle-ci doit pointer vers la cellule courante (*ref*).

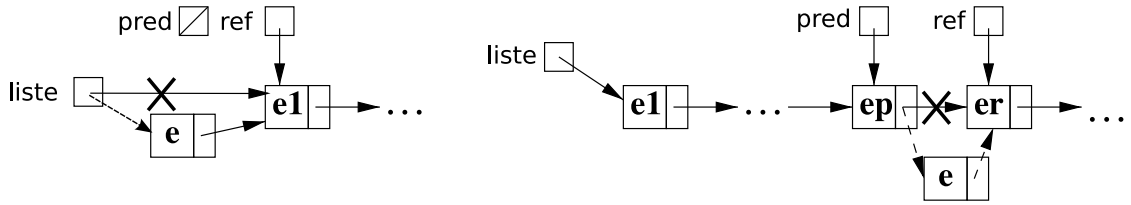

FIGURE 14.5 – Insertion dans une liste triée

Dans la figure 14.5, à droite, la référence *ref* peut ne pas pointer vers une cellule. Ceci arrive quand l'élément à insérer est plus grand que tous les éléments de la liste. Dans ce cas, l'insertion se fait à la fin de la liste, donc au moment de l'arrêt de la recherche, *ref* est nulle et *pred* est sur la dernière cellule de la liste. Ce cas est identique à celui présenté dans la figure 14.5, à la variable `suivant` de la nouvelle cellule on donne tout simplement la valeur de *ref*.

Listing 14.22 – (lien vers le code brut)

```
1   public void ajouterTriee(int v) {
2     if (estVide()) {
3       premier = new ElementListe(v);
4     } else if (getPremier().getValeur() >= v) {
5       // Insertion au début de la liste.
6       premier = new ElementListe(v, premier);
7     } else {
8       // On veut chercher l'élément qui précède notre valeur
9       // Le problème est que nous ne savons que nous avons
10      // l'élément ``précédent'' que parce que le ``suivant''
11      // est strictement plus grand que v.
12      ElementListe precedent = getPremier(); // Initialisation correcte car
13                                             // getPremier().getValeur() < v.
14      ElementListe elt = getPremier().getSuivant();
15      while (elt != null && elt.getValeur() < v) {
16        precedent = elt;
17        elt = elt.getSuivant();
18      }
19      precedent.setSuivant(new ElementListe(v, elt));
20    }
21  }
```

Variante récursive :

Nous utilisons le même schéma de décomposition récursive de la liste, en un premier élément (*premier*) et un reste de la liste (*reste*). L'insertion récursive suit le schéma suivant :

```
liste.insertionTriee(e) =
    ListeTriee(e, liste)                        si e<=premier
    liste après reste=ListeTriee(e, null)       si e>premier et reste==null
    liste après reste=reste.insertionTriee(e)   si e>premier et reste!=null
```

Si l'élément à insérer est plus petit ou égal à *premier*, l'insertion se fera en tête de liste. Sinon, l'insertion se fera récursivement dans le reste. Si *reste* est vide, alors *reste* sera remplacé par la nouvelle cellule à insérer. Si *reste* n'est pas vide, l'insertion se fait récursivement dans *reste* et le résultat remplace l'ancien *reste*.

Listing 14.23 – (lien vers le code brut)

```
1   public void ajouterTriee(int v) {
2     premier= ajouterTrieeRec(premier,v);
3   }
4
5   private ElementListe ajouterTrieeRec(ElementListe elt, int v) {
6     if (elt == null) {
7       return new ElementListe(v);
8     } else if (elt.getValeur() > v) {
9       return new ElementListe(v, elt);
10    } else {
11      ElementListe e= ajouterTrieeRec(elt.getSuivant(), v);
12      elt.setSuivant(e);
13      return elt;
```

```
14      }
15    }
```

14.4.3 Suppression dans une liste triée

La suppression de la première occurrence d'un élément dans une liste triée est assez similaire à la suppression dans une liste non triée. La seule chose qui change est la recherche de la cellule à supprimer, qui bénéficie du tri des valeurs. Aussi, si plusieurs occurrences de l'élément existent, elles sont forcément l'une à la suite de l'autre à cause du tri. Il serait donc plus simple d'éliminer *toutes* les occurrences de l'élément que dans le cas des listes non triées. Nous nous limiterons cependant ici à la suppression de la première occurrence seulement.

Variante iterative :

Par rapport à la méthode `suppressionPremier` de la classe `Liste`, ici le parcours de la liste à la recherche de l'élément ne se fait que tant que *ref* pointe une valeur plus petite que celui-ci. Une fois le parcours arrêté, la seule différence est qu'il faut vérifier que *ref* pointe vraiment la valeur recherchée.

Listing 14.24 – (lien vers le code brut)

```
1   public void retirerPremiereOccurrence(int v) {
2     // On élimine le problème de la liste vide
3     if (estVide())
4       return;
5     // Le but est de trouver l'élément qui précède v...
6     // qui n'existe pas si v est la première valeur=>
7     if (premier.getValeur() == v) {
8       premier = premier.getSuivant();
9     } else {
10      ElementListe precedent = null;
11      ElementListe elt = premier;
12      while (elt != null && elt.getValeur() < v) {
13        precedent = elt;
14        elt = elt.getSuivant();
15      }
16      if (elt != null && elt.getValeur() == v) {
17        // L'élément a été trouvé
18        // Plomberie:
19        precedent.setSuivant(elt.getSuivant());
20      }
21    }
22  }
```

Variante récursive :

Par rapport à la variante récursive de la méthode `suppressionPremier` de la classe `Liste`, une nouvelle condition d'arrêt est rajoutée :
quand l'élément à éliminer est plus petit que *premier*, il n'existe sûrement pas dans la liste et celle-ci est retournée non modifiée.

Listing 14.25 – (lien vers le code brut)

```
1   public void retirerPremiereOccurrence(int v) {
2     premier= retirerPremiereOccurrenceRec(premier, v);
3   }
4
5   private ElementListe retirerPremiereOccurrenceRec(ElementListe elt, int v) {
6     if (elt == null) {
7       return null;
8     } else if (v == elt.getValeur()) {
9       return elt.getSuivant();
10    } else if (v < elt.getValeur()) {
11      return elt;
12    } else {
13      elt.setSuivant(retirerPremiereOccurrenceRec(elt.getSuivant(),v));
14      return elt;
15    }
16  }
```

14.5 Un exemple de programme qui utilise les listes

Considérons un exemple simple de programme qui manipule des listes chaînées en utilisant la classe `Liste`.

Le programme lit une suite de nombres entiers terminée par la valeur 0 et construit une liste avec ces entiers (sauf le 0 final). La liste doit garder les éléments dans l'ordre dans lequel ils sont introduits. Egalement, une valeur ne doit être stockée qu'une seule fois dans la liste.

Le programme lit ensuite une autre suite de valeurs, également terminée par un 0, avec lesquelles il construit une autre liste, sans se soucier de l'ordre des éléments. Les éléments de cette seconde liste devront être éliminés de la liste initiale.

A la fin, le programme affiche ce qui reste de la première liste.

Remarque : Pour garder les éléments dans l'ordre d'introduction, l'insertion d'un nouvel élément doit se faire *en fin de liste*. La méthode `ajouterAuDebut` ne convient donc pas. On pourrait se débrouiller avec la méthode `concatener`, mais comme l'opération est fréquemment utile, on suppose écrite la méthode `ajouterALaFin`.

Remarque : Pour qu'un élément soit stocké une seule fois dans la liste, il faut d'abord vérifier s'il n'y est pas déjà, à l'aide de la méthode `contient`.

Voici le programme qui réalise ces actions :

Listing 14.26 – (lien vers le code brut)

```
1  public class ExempleListes{
2    public static void main(String[] args){
3      //1. Creation premiere liste
4      Terminal.ecrireStringln("Entrez les valeurs terminees par un 0 :");
5      Liste liste = new Liste();  //liste a construire
6      do{
7        int val = Terminal.lireInt();
8        if(val==0) break;
9        if (! liste.contient(val))
10         liste.ajouterALaFin(val);
11     } while(true);
```

```
12
13      //2. Creation seconde liste
14      Terminal.ecrireStringln("Valeurs a eliminer (terminees par un 0) :");
15      Liste liste2 = new Liste();
16      do{
17        int val = Terminal.lireInt();
18        if(val==0) break;
19        liste2.ajouterAuDebut(val);
20      } while(true);
21
22      //3. Elimination des éléments de la premiere liste
23      for(ElementListe ref = liste2.getPremier(); ref != null; ref = ref.getSuivant()){
24        if(liste.estVide()) break;  //plus rien a eliminer
25        liste.retirerPremiereOccurrence(ref.getValeur());
26      }
27
28      //4. Affichage liste restante
29      Terminal.ecrireStringln("Valeurs restantes :");
30      for(ElementListe ref = liste.getPremier(); ref != null; ref = ref.getSuivant())
31        Terminal.ecrireString("" + ref.getValeur());
32      Terminal.sautDeLigne();
33    }
34  }
```